Franklin Content Management Prototype

Documentation

IBM Confidential

IBM Advanced Internet Technology Group (WebAhead)
For more information, contact
Sara Elo (saraelo@us.ibm.com) or
Dikran Meliksetian (meliksd1@us.ibm.com)

Franklin team members:
Peter Davis
Sara Elo
Abel Henry
Dikran Meliksetian
Jeff Milton
Louis Weitzman
Jessica Wu
Joe Zhou

A

Table of Contents

BLANK PAGE

Overview

Content on the Next Generation internet needs to be highly adaptive. New interfaces and devices are emerging, the diversity of users is increasing, machines are acting more and more on users' behalf, and net activities are possible for a wide range of business, leisure, education, and research activities.

To achieve maximum flexibility and reuse, content needs to be broken down into richly tagged fragments that can be combined and rendered appropriately for the user, task, and context. The Franklin content management prototype builds on this premise. It provides an end-to-end process from content creation and meta-tagging to quality assurance and publishing.

Franklin integrates several IBM technologies for its five components: content store, meta-data store, dispatcher, services and user interfaces. A high-level view of the components is shown in Figure 1: Franklin Components.

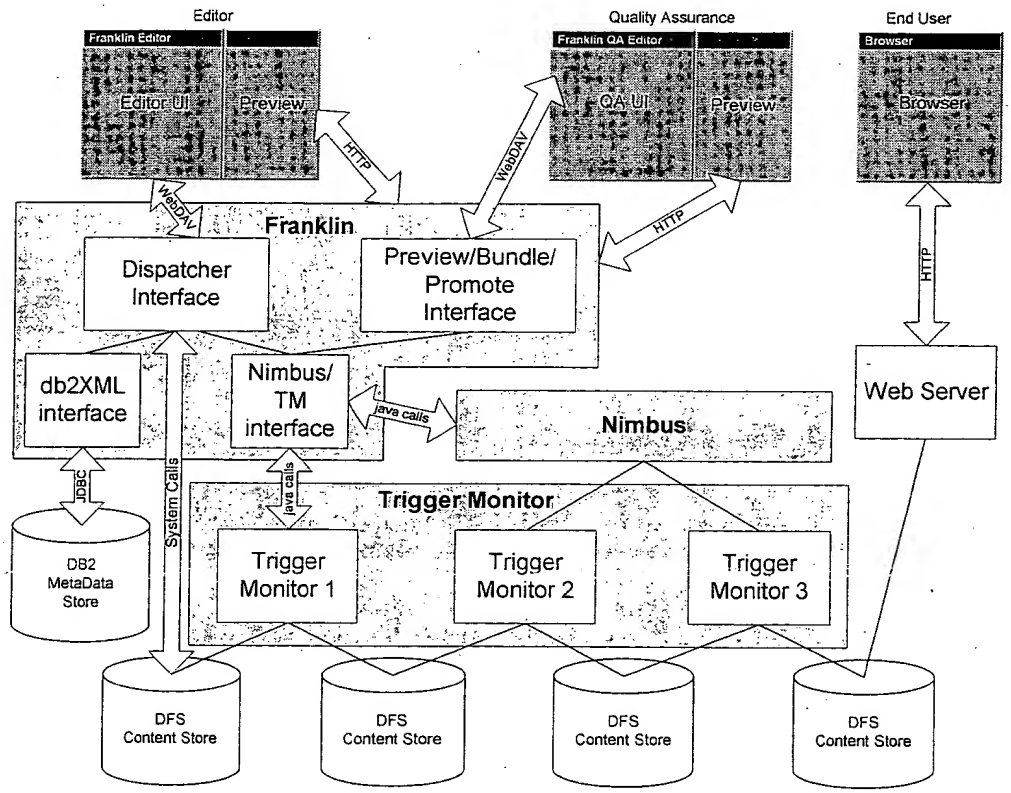


Figure 1: Franklin Components

The content store builds upon the Daedalus (a.k.a Trigger Monitor) technology from IBM Watson Research. [For full specification, see http://w3.watson.ibm.com/~challngr/papers/daedalus/index.html] Daedalus is designed to manage high numbers of rapidly changing content fragments. By maintaining an Object Dependency Graph, and by detecting changes to content, it manages pages on a web server or cached in a network router in a timely manner.

The meta-data store manages tags that describe the functional and semantic role of each content fragment within the information collection. They may describe what the content is about, who the target audience is, and its relationship to a taxonomy or other fragments. The meta-data store also supports efficient searches.

Networked services support the editor in content creation. They may assist the editor in meta-data creation, classification, summarization or translation. Instead of doing the task from beginning to end, the editor can accept, reject or modify the suggestions created by a service.

The dispatcher's task is to delegate incoming requests to the content store, meta-data store and the services. The dispatcher presents a consistent application programming interface to the user interfaces. This Franklin API abides to the Web protocol for Distributed Authoring and Versioning (WebDAV) and to the Distributed Authoring Search Language (DASL) specification.

The user interfaces communicate with the Franklin system through the API. Using the Editor UI, an editor can create and edit XML content fragments, upload XSL style sheets and multimedia objects, compose pages out of fragments, preview pages, review final published pages, and reject them or promote them to the final stage in the publishing flow.
The Franklin system has also been integrated with KittyHawk, an IBM Notes based workflow engine. This workflow module can be turned on or off depending on the application needs.

This document describes in detail the system requirements and setup, the architecture, components and features of Franklin. It covers the lessons learned, and provides a working example of a content collection managed with Franklin. In addition, it describes a lightweight version of Franklin, code-named Franklin Light, which satisfies the needs of small sites with no need for multiple Quality Assurance steps, or a scalable DB2 based search.

System Setup & Configuration

Before running an instance of Franklin to manage the content for a web site, you need to complete a number of installation steps. You also need to define the DTDs, the XSL style sheets, and the site map of the web site you intend to manage. This section outlines the required steps.

Step 1: Install Franklin Server

The Franklin Server runs on an AIX or NT server and requires the following software installed on the same machine:

- Apache Web Server v.1.3.6 or higher
- WebSphere Application Server v.2.0 or higher
- Java run-time environment 1.1.8

The Franklin server is distributed as a jar file, i.e., Franklin.jar. The distribution directory contains the following jar files that are required by Franklin: xml4j.jar, patbin132.jar, daedalus.jar, lotusxsl.jar, xerces.jar.

Deleted:

Download the Franklin Server and associated components from http://franklin.adtech.internet.ibm.com/franklin/downloads/index.html and place them in a directory accessible by the WebSphere Application Server.

Step 2: Install DB2 for Meta-Data Store

The DB2 database used by the Meta-data Store can run on the same machine or a different machine. It requires the following software:

1) DB2 6.1 with DB2 XML Extender 7.1
   Download DB2 XML Extenders 7.1 from IBM software website at http://www.software.ibm.com/db2. Currently, XML Extenders is supported on Windows NT, AIX and Solaris. If you decide to use the XML Extender Administration Wizard make sure you review the XML Extender Administration Wizard Readme file to ensure you have the software prerequisites, JDK 1.1.x or JRE v1.1.x and JFC 1.1 with Swing 1.1 or later.
2) JDBC for DB2 JDBC 1.20
   JDBC is included in the DB2 installation (db2java.zip) in the directory of sqllib/java.

The steps to install DB2 and enable DB2 XML Extenders (which require root authority on AIX):

1) Install a version of UDB higher than 5.2. We have tested DB2 XML on NT for UDB 5.2 and 6.1, and on AIX for UDB 6.1 for DB2 XML XColumn function.
2) Create a DB2 instance. In the included examples, we use the db2 instance name db2frnkl.
3) Install DB2 XML Extenders
4) Create a database in the instance. Also, create the tables and indexes based on the sample scripts we have provided.
5) Enable the database with XML Extenders
6) Start JDBC on a port. For example, "db2jstrt 4000" opens port 4000 for JDBC connections.

Step 3: Customize Server Initialization Files

Edit *franklinServletInitialization.properties* file and set the following variable to the desired directory in your setup:

baseDir – base directory for all Franklin related files.

Comment [LW1]: Should we name the properties files differently? E.g. franklinClient.properties vs franklinServer.properties?

All other variables in *franklinServletInitialization.properties* are relative to *baseDir* and should not be changed:

dtdDir - directory for DTD and entity files
xmlDir - root of the directory hierarchy for XML files

assetsDir = directory for all directories browsable by client UI, i.e. *xslDir, publishDir, multimediaDir*
xslDir - directory for XSL style sheets
publishDir - root of the directory hierarchy for HTML, HDML or DHTML files
multimediaDir - root of the directory hierarchy for images, graphics, video and audio files

Edit *metastore.ini* file and set the following variables to the desired directory in your setup:

MetaStoreServerIP - database host machine name
MetaStoreServerPort - JDBC port number
MetaStoreServerDBName – database name
MetaStoreServerUserID - database user name
MetaStoreServerPassword - database password for the above user
MetaStoreServerDriverClassName - database JDBC driver name
MetaStoreServerInitialConnection - number of initial connections to the database
MetaStoreServerIncrement - number of additional connections to database
MetaStoreCheckInXMLDir - temporary directory for XML files checked into meta store
MetaStoreDADDir – directory for DAD files
MetaStoreCacheSearchDir - directory for cached XML Search results

Step 4: Configure WebSphere Application Server

Start the WebSphere Administrative Console, refer to the WebSphere Quick Beginnings guide for details

1. In the Tasks tab of the console, select Configure a Web Application and click the start task button
   a. Specify the Web Application Name, e.g., FranklinServer, click Next.
   b. Choose the servlet engine, e.g., the ServletEngine in the Default Server of the Default Host, click Next
   c. Specify the Web Application Web Path, e.g., /franklinserver, click Next
   d. Specify the CLASSPATH:
      i. Add each of the jar files in the Franklin distribution to the classpath, i.e., franklin.jar, daedalus.jar, xml4j.jar, lotusxsl.jar, xerces.jar, patbin132.zip
      ii. Add the db2java.zip file to the classpath, the db2java.zip file is distributed with DB2, it is found under the sqllib/java subdirectory of the database instance home directory
      iii. Click Finished
2. In the same Tasks tab, select Add a Servlet and click the start task button
   a. Select Yes to "Do you want to select an existing Servlet jar file or Directory that contains Servlet classes" and click Next
   b. Specify the path of the directory where franklin.jar is located, click Next
   c. Select the Web Application that was created in the previous step, click Next
   d. Select the Create User-Defined Servlet option, click Next
   e. Specify the Servlet Name, e.g., dispatcher
   f. Specify the Servlet Class Name as com.ibm.adtech.franklin.server.dispatcher.Dispatcher

g. Specify the Servlet Web Path List, for example /franklinserver/dispatcher, this is the web path that should be used by the client to access the franklin server, click next
h. Add an init parameter with Init Parm Name as baseDir and Init Parm Value equal to the directory where the franklin server configuration files are stored, e.g., /franklin/data/config
i. Select the True option for Load at Startup; click Finished

3. The configuration is complete you need to start the service, select the Topology tab
   a. Prior to starting the application, make sure that the database instance is running and that jdbc daemon is active (see previous section)
   b. Expand the topology tree and select the newly created application. The application will appear under the servlet engine that was selected in step 1.b
   c. Right click the selection and on the popup menu select Restart Application
4. The Franklin Server should be available at this point. In order to verify that everything is in order view the log files of WebSphere

Step 5: Install Franklin Client

The Franklin Client Java Application has been tested on Windows98/2000/NT.

Download the Franklin Client Application Installer FranklinEditor.exe from http://franklin.adtech.internet.ibm.com/franklin/downloads/index.html and run it. In addition to the Franklin Client, the following Java packages are required and are automatically installed by the Installer:

- Java 1.1.8 run-time environment with Swing JFC1.1.1
- XML4J package
- WebDav package

The Franklin Client Application Installer also creates the subdirectories required by the client under the chosen installation directory. You can change these directories as described in the next paragraph.

After installation, customize the initialization file *franklin.properties* located in the root directory where you installed the Franklin Client application. You need to edit the variable *browserPath* to define the location of the web browser you wish to use to preview pages. Also, you can edit the variable *tempDir* if you wish to change the directory where temporary files are stored.

Comment [LW2]: Should variable names in the client properties file be more similar to the names of the variables in the dispatcher's properties file?

```
dispatcher      = http://adtech.ibmus2.ibm.com/franklinservlet/
initXMLFile     = xml/franklin_init.xml
## modify browserPath to point to the web browser you wish to use for preview
browserPath     = c:/Program Files/Internet Explorer/iexplore.exe
## modify tempDir to point to the directory where temporary files will be stored
tempDir         = ./tmp/
tempMediaDir    = media/
tempHTMLDir     = html/
tempXSLDir      = xsl/
standaloneP     = false
validateP       = true
```

Step 6: Define Document Type Definitions (DTD)

Franklin manages two types of content objects, *fragments* and *servables*.

A fragment is a content object that can be reused on several pages:

- a *simple fragment* is a self contained XML file containing text data and metadata – for example, a product specification
- a *compound fragment* is an XML file that contains metadata and points to an accompanying file such as a video or image file, an XSL style sheet, or a hand-crafted HTML page
- an *index fragment* is an automatically updated XML file that indexes any number of servables - for example a panel listing the five latest press release [**Future**: index fragments not available in current implementation]

A servable is an XML file that contains the text and meta-data for one final published page and imports reusable content from one or more fragments, and points to one of more style sheet fragments.

Figure 2 shows a product page servable which includes content from six fragments, namely three text fragments, one image fragment and two style sheet fragments, and results in two final published pages.

Insert Figure 2 here

Before beginning to manage a content collection, you need to define the document type definitions, or DTDs, for each class of fragment and servable that will be managed by the application. Franklin uses the syntax of DTDs to define a document type. [See the XML specification at http://www.w3.org/TR/REC-xml]

In order for Franklin to manage DTDs correctly, all DTDs must abide to the Franklin following specifications:

Franklin specification of fragment and servable DTDs

1. The root element, to which you can give a meaningful name, must have a child node called `SYSTEM` with the children nodes `FRAGMENTID, CREATOR, MODIFIER, CREATIONTIME, LASTMODIFIEDTIME, PAGETYPE` and `CONTENTSIZE`. The `NAME` attribute of `PAGETYPE` must be set to either `"FRAGMENT"` or `"SERVABLE"`.

```
<!ELEMENT ROOT          (SYSTEM, …)>

<!ELEMENT SYSTEM        (FRAGMENTID,
                        CREATIONTIME,
                        LASTMODIFIEDTIME,
                        CREATOR,
                        MODIFIER,
                        PAGETYPE
                        CONTENTSIZE?)>
```

```
<!ELEMENT FRAGMENTID        (#PCDATA)>
<!ELEMENT CREATIONTIME      (#PCDATA)>
<!ELEMENT LASTMODIFIEDTIME  (#PCDATA)>
<!ELEMENT CREATOR           (#PCDATA)>
<!ELEMENT MODIFIER          (#PCDATA)>
<!ELEMENT CONTENTSIZE       (#PCDATA)>
<!ELEMENT PAGETYPE          (#PCDATA)>
<!ATTLIST PAGETYPE          NAME (FRAGMENT|SERVABLE) "FRAGMENT" #FIXED>
```

2. All items editable in the Editor UI need to be elements of the DTD, not attributes. For example,

```
<!ELEMENT TITLE             (#PCDATA)>
<!ELEMENT SHORTDESCRIPTION  (#PCDATA)>
<!ELEMENT CATEGORY          (#PCDATA)>
```

3. All elements to be indexed for search must be of type PCDATA, and must contain the attribute SEARCH set to YES. For example,

```
<!ELEMENT ROOT              (TITLE, SHORTDESCRIPTION, CATEGORY, ...)>
<!ELEMENT TITLE             (#PCDATA)>
<!ELEMENT SHORTDESCRIPTION  (#PCDATA)>
<!ELEMENT CATEGORY          (#PCDATA)>
<!ATTLIST TITLE             SEARCH (YES|NO) "YES" #FIXED>
<!ATTLIST SHORTDESCRIPTION  SEARCH (YES|NO) "YES" #FIXED>
<!ATTLIST CATEGORY          SEARCH (YES|NO) "YES" #FIXED>
```

Future: Need to add SEARCH attribute. The SEARCH attribute will allow Franklin to automatically generate the DAD mapping for the DB2 XML Extenders..

4. Include the external entity reference that defines the user interface widgets recognized by the Franklin Editor UI. Each element that needs to be editable in the Editor UI must be of type PCDATA and contain the DATATYPE attribute set to the appropriate UI type.

```
<!ENTITY  % UITYPES       SYSTEM
"http://franklinserver/franklin/dtd/uitypes.txt">

<!ATTLIST TITLE             DATATYPE (%UITYPES;)       "STRING"    #FIXED>
<!ATTLIST SHORTDESCRIPTION  DATAYTPE (%UITYPES;)       "LONGTEXT"  #FIXED
                            PARSE    (TRUE)            "TRUE"      #FIXED>
```

If you wish a LONGTEXT widget to allow an editor to enter a limited set of HTML tags, add the PARSE attribute and set it to true. The supported HTML are:

```
<p>, <ul>, <ol>, <sl>, <dl>, <dt>, <dd>, <li>, <div>
```

The file *uitypes.txt* is fixed and provided in the Franklin install in the *dtdDir* in the *franklin.properties* file. It contains the list of all UI widgets known to the Editor UI. (See section Editor UI Widgets for a detailed description of UITYPES)

```
DATE | INTEGER | STRING | SHORTTEXT | LONGTEXT | CHOICE | BROWSESERVER |
BROWSELOCAL | ASSOCLIST
```

5. An element can appear as a drop-down menu in the Editor UI and restrict the editor to choose the value from a predefined set. To accomplish this, set the DATATYPE attribute to the UITYPE "CHOICE" and the CHOICES attribute to a default value from a list of options. The options can be defined as an external entity for reuse across many DTDs.

```
<!ENTITY  % CATEGORYDEFS       SYSTEM
"http://franklinserver/franklin/dtd/categorydefs.txt">
<!ATTLIST CATEGORY          DATATYPE (%UITYPES;)       "CHOICE"     #FIXED
                            CHOICES  (%CATEGORYDEFS;) "NONE"       #REQUIRED>
```

For example, the options for `CATEGORY` could be defined as the types of Netfinity servers:

```
NONE | Netfinity_8500R | Netfinity_7000_M10 | Netfinity_5500_M10 |
Netfinity_5600 | Netfinity_5500
```

The Editor UI assumes that if the first word in the set of CHOICES is the string NONE, and the editor selects it, the element will not appear in the XML document.

6. A fragment can include other fragments as subfragments. If so, the entity reference that defines all subfragment types must be included in the DTD. The declaration of a subfragment must contain the `SUBFRAGMENTTYPE` attribute set to the appropriate type.

```
<!ENTITY % SUBFRAGMENTTYPES SYSTEM
"http://franklinserver/franklin/dtd/subfragmenttypes.txt">
<!ELEMENT SUBFRAGMENT (#PCDATA)>
<!ATTLIST SUBFRAGMENT SUBFRAGMENTTYPE (%SUBFRAGMENTTYPES;) "IMAGEFRAGMENT"
#FIXED>
```

Future: the subfragment syntax will be replaced by the XLink syntax once it becomes a W3 recommendation and XML4J and LotusXSL support the syntax. Until then, we will use subfragment elements as way to include content from another fragment.

An example of a fragment DTD, listfragment.dtd:

```
<!ENTITY % SUBFRAGMENTTYPES SYSTEM
"http://franklinserver/franklin/dtd/subfragmenttypes.txt">
<!ENTITY  % CATEGORYDEFS       SYSTEM
"http://franklinserver/franklin/dtd/categorydefs.txt">
<!ENTITY  % UITYPES      SYSTEM
"http://franklinserver/franklin/dtd/uitypes.txt">

<!ELEMENT LISTFRAGMENT (SYSTEM, TITLE, SHORTDESCRIPTION?, CATEGORY*,
                        LISTITEM+)>
<!ELEMENT SYSTEM       (FRAGMENTID, CREATOR, MODIFIER, CREATIONTIME,
                        LASTMODIFIEDTIME, PAGETYPE, CONTENTSIZE?)>
<!ELEMENT FRAGMENTID        (#PCDATA)>
<!ELEMENT CREATIONTIME      (#PCDATA)>
<!ELEMENT LASTMODIFIEDTIME  (#PCDATA)>
<!ELEMENT CONTENTSIZE       (#PCDATA)>
<!ELEMENT CREATOR           (#PCDATA)>
<!ELEMENT MODIFIER          (#PCDATA)>
<!ELEMENT PAGETYPE         (#PCDATA)>
<!ELEMENT TITLE             (#PCDATA)>
<!ELEMENT SHORTDESCRIPTION  (#PCDATA)>
```

```
<!ELEMENT CATEGORY          (#PCDATA)>
<!ELEMENT LISTITEM          (LISTTITLE?, DESCRIPTION?, LINK?, FOOTNOTE?)>
<!ELEMENT LISTTITLE         (#PCDATA)>
<!ELEMENT DESCRIPTION       (#PCDATA)>
<!ELEMENT LINK              (#PCDATA)>
<!ELEMENT FOOTNOTE          (#PCDATA)>

<!ATTLIST TITLE             DATATYPE (%UITYPES;)     #FIXED       "STRING"
                            SEARCH   (YES|NO) "YES" #FIXED>

<!ATTLIST SHORTDESCRIPTION DATATYPE (%UITYPES;)     #FIXED       "SHORTTEXT"
                            SEARCH   (YES|NO) "YES" #FIXED>
<!ATTLIST CATEGORY          DATATYPE (%UITYPES;)       #FIXED     "CHOICE"
                            CHOICES  (%CATEGORYDEFS;) #IMPLIED
                            SEARCH   (YES|NO) "YES" #FIXED>
<!ATTLIST LISTTITLE         DATATYPE (%UITYPES;)     #FIXED       "STRING">
<!ATTLIST DESCRIPTION       DATATYPE (%UITYPES;)     #FIXED       "STRING">
<!ATTLIST LINK              DATATYPE  (%UITYPES;)    #FIXED       "STRING">
<!ATTLIST FOOTNOTE          DATATYPE  (%UITYPES;)    #FIXED       "STRING">
```

Franklin specification of compound fragment DTDs

A compound fragment contains a pointer to an accompanying file, such as a multimedia file, an XSL style sheet or a hand-crafted HTML file (i.e. ones that are not generated from XML by Franklin) The accompanying file is encoded as a binary object into the fragment for the duration of the communication between Editor UI and Franklin Server. Before check-in to the server, the Editor UI encodes the file as a binary object into the XML fragment. At the receiving end, the dispatcher extracts it and decodes into the original format. The reverse happens at check-out. This allows a single communication between the client and server when exchanging this type of fragments.

1. A compound fragment DTD must use the following syntax to declare the inclusion of an external file:

```
<!ELEMENT CONTENT           (#PCDATA)>
<!ELEMENT CONTENTDIR        (#PCDATA)>
<!ELEMENT CONTENTFILENAME   (#PCDATA)>
<!ATTLIST CONTENT          DATATYPE (%UITYPES;)  #FIXED      "STRING">
<!ATTLIST CONTENTDIR       DATATYPE (%UITYPES;)  #FIXED      "BROWSESERVER">
<!ATTLIST CONTENTFILENAME DATATYPE (%UITYPES;)  #FIXED      "BROWSELOCAL">
```

Franklin specification of group index fragment DTDs

Future: To be filled in

Franklin specification of servable DTDs

In the Franklin system, servables always result in one of more final published pages. The DTD must indicate the names of the XSL style sheets it can use for layout and where to publish the resulting pages.

1. A servable DTD must contain the following declarations:

```
<!ELEMENT PUBLISHINFO       (STYLESHEET, PUBLISHDIR, PUBLISHFILENAME)>
<!ELEMENT STYLESHEET        (#PCDATA)>
<!ELEMENT PUBLISHDIR        (#PCDATA)>
<!ELEMENT PUBLISHFILENAME   (#PCDATA)>
<!ATTLIST STYLESHEET       DATATYPE (%UITYPES;) #FIXED "CHOICE"
          CHOICES (styleshheet1.xsl|stylesheet2.xsl) #IMPLIED>
<!ATTLIST PUBLISHDIR      DATATYPE  (%UITYPES;) #FIXED "BROWSESERVER">
<!ATTLIST PUBLISHFILENAME DATATYPE  (%UITYPES;) #FIXED "STRING">
```

2. A servable can include one or more subfragments. Each subfragment serves a specific role within the servable and can be named in a meaningful way, for example MAINPHOTO, HIGHLIGHTS etc. Each subfragment must have an attribute that indicates the type of subfragment to include. The syntax to include a subfragment in a servable follows:

```
<!ELEMENT MAINPHOTO         (#PCDATA)>
<!ELEMENT HIGHLIGHTS        (#PCDATA)>
<!ATTLIST MAINPHOTO       DATATYPE (%UITYPES;)   #FIXED "STRING"
          SUBFRAGMENTTYPE (%SUBFRAGMENTTYPES;)   #FIXED "IMAGEFRAGMENT">
<!ATTLIST HIGHLIGHTS      DATATYPE (%UITYPES;)   #FIXED "STRING"
          SUBFRAGMENTTYPE (%SUBFRAGMENTTYPES;)   #FIXED "LISTFRAGMENT">
```

3. A servable can be included in an automatically generated group index fragment.

To be filled in

An example of a servable DTD, productpage.dtd:

```
<!ENTITY % UNIVERSAL        SYSTEM
"http://franklinserver/franklin/dtd/universal.dtd">
<!ENTITY % SUBFRAGMENTTYPES SYSTEM
"http://franklinserver/franklin/dtd/subfragmenttypes.txt">
<!ENTITY  % CATEGORYDEFS     SYSTEM
"http://franklinserver/franklin/dtd/categorydefs.txt">
<!ELEMENT PRODUCTPAGE (SYSTEM, TITLE, SOURCE?, COMMENT?, SHORTDESCRIPTION?,
                       LONGDESCRIPTION?, KEYWORD*, CATEGORY*, RELATEDLINK*,
                       PUBLISHINFO+, BRANDNAVIGATION, MAINPHOTO, GLANCE,
                       HIGHLIGHTS , GROUPINDEX*)>
<!ELEMENT SYSTEM     (FRAGMENTID, CREATOR, MODIFIER, CREATIONTIME,
                       LASTMODIFIEDTIME, PAGETYPE, CONTENTSIZE?)>
<!ELEMENT FRAGMENTID        (#PCDATA)>
<!ELEMENT CREATIONTIME      (#PCDATA)>
<!ELEMENT LASTMODIFIEDTIME  (#PCDATA)>
<!ELEMENT CONTENTSIZE       (#PCDATA)>
<!ELEMENT CREATOR           (#PCDATA)>
<!ELEMENT MODIFIER          (#PCDATA)>
<!ELEMENT PAGETYPE          (#PCDATA)>
<!ELEMENT TITLE             (#PCDATA)>
<!ELEMENT SOURCE            (#PCDATA)>
<!ELEMENT COMMENT           (#PCDATA)>
<!ELEMENT SHORTDESCRIPTION  (#PCDATA)>
<!ELEMENT LONGDESCRIPTION   (#PCDATA)>
<!ELEMENT KEYWORD           (#PCDATA)>
```

```
<!ELEMENT CATEGORY          (#PCDATA)>
<!ELEMENT RELATEDLINK       (URL, LINKTITLE)>
<!ELEMENT URL               (#PCDATA)>
<!ELEMENT LINKTITLE         (#PCDATA)>
<!ELEMENT DOCTYPE           (#PCDATA)>
<!ELEMENT DTDURL            (#PCDATA)>
<!ELEMENT PUBLISHINFO  (STYLESHEET, PUBLISHDIR, PUBLISHFILENAME)>
<!ELEMENT STYLESHEET        (#PCDATA)>
<!ELEMENT PUBLISHDIR        (#PCDATA)>
<!ELEMENT PUBLISHFILENAME   (#PCDATA)>
<!ELEMENT BRANDNAVIGATION   (#PCDATA)>
<!ELEMENT MAINPHOTO         (#PCDATA)>
<!ELEMENT GLANCE            (#PCDATA)>
<!ELEMENT HIGHLIGHTS        (#PCDATA)>
<!ELEMENT GROUPINDEX        (#PCDATA)>
<!ATTLIST TITLE             DATATYPE (%UITYPES;)       #FIXED       "STRING">
<!ATTLIST SOURCE            DATATYPE (%UITYPES;)       #FIXED       "STRING">
<!ATTLIST COMMENT           DATATYPE (%UITYPES;)     FIXED       "STRING">
<!ATTLIST SHORTDESCRIPTION DATATYPE (%UITYPES;)     #FIXED       "SHORTTEXT">
<!ATTLIST LONGDESCRIPTION   DATATYPE (%UITYPES;)       #FIXED
"SHORTTEXT">
<!ATTLIST KEYWORD           DATATYPE (%UITYPES;)       #FIXED       "STRING">
<!ATTLIST CATEGORY          DATATYPE (%UITYPES;)       #FIXED       "CHOICE"
                            CHOICES  (%CATEGORYDEFS;) #IMPLIED>
<!ATTLIST URL               DATATYPE (%UITYPES;)       #FIXED       "STRING">
<!ATTLIST LINKTITLE         DATATYPE (%UITYPES;)       #FIXED       "STRING">
<!ATTLIST BRANDNAVIGATION   DATATYPE (%UITYPES;)       #FIXED     "STRING"
          SUBFRAGMENTTYPE (%SUBFRAGMENTTYPES;)  #FIXED LISTFRAGMENT"
<!ATTLIST MAINPHOTO      DATATYPE  (%UITYPES;)         #FIXED     "STRING"
          SUBFRAGMENTTYPE (%SUBFRAGMENTTYPES;)  #FIXED "IMAGEFRAGMENT">
<!ATTLIST GLANCE         DATATYPE   (%UITYPES;)       #FIXED      "STRING"
          SUBFRAGMENTTYPE (%SUBFRAGMENTTYPES;)  #FIXED "LISTFRAGMENT">
<!ATTLIST HIGHLIGHTS     DATATYPE   (%UITYPES;)  #FIXED    "STRING"
          SUBFRAGMENTTYPE (%SUBFRAGMENTTYPES;)  #FIXED "LISTFRAGMENT">
<!ATTLIST STYLESHEET     DATATYPE   (%UITYPES;) #FIXED "CHOICE"
          CHOICES  (web_product_index.xsl | pda_product_index.xsl)
           #IMPLIED>
<!ATTLIST PUBLISHDIR     DATATYPE  (%UITYPES;)  #FIXED "BROWSESERVER">
<!ATTLIST PUBLISHFILENAME DATATYPE(%UITYPES;)  #FIXED "STRING">
<!ATTLIST GROUPINDEX      DATATYPE (%UITYPES;)  #FIXED "STRING"
          SUBFRAGMENTTYPE (%SUBFRAGMENTTYPES;)  #FIXED "GROUPINDEX">
```

An example of a servable, 2-tsrv.xml, which abides to productpage.dtd:

```
<?xml version="1.0"?>
<!DOCTYPE PRODUCTPAGE SYSTEM "http://franklinserver/dtd/productpage.dtd">
<PRODUCTPAGE>
   <SYSTEM>
      <FRAGMENTID>2-tsrv.xml</FRAGMENTID>
      <CREATOR>Joe Moe</CREATOR>
      <MODIFIER>Jane Mane</MODIFIER>
      <CREATIONTIME>384738740383</CREATIONTIME>
      <LASTMODIFIEDTIME>384738740383</LASTMODIFIEDTIME>
      <PAGETYPE>Servable</PAGETYPE>
   </SYSTEM>
   <TITLE>Netfinity 8500R</TITLE>
```

```
    <SOURCE>IBM PC Company</SOURCE>
    <SHORTDESCRIPTION>Mainframe features bring extraordinary performance and
 reliability to a rack-optimized 8-way server</SHORTDESCRIPTION>
   <LONGDESCRIPTION>A great value in 8-way servers, the new Netfinity 8500R
 maximizes uptime and provides superior manageability for compute-intensive
 business intelligence, transaction processing and server consolidation
 projects. </LONGDESCRIPTION>
    <KEYWORD>New</KEYWORD>
    <KEYWORD>Server</KEYWORD>
    <KEYWORD>Pentium</KEYWORD>
    <CATEGORY>Netfinity 8500R</CATEGORY>
    <CATEGORY>Small and Medium Business</CATEGORY>
    <RELATEDLINK>
       <URL>ftp://ftp.pc.ibm.com/pub/pccbbs/pc_servers/8500rf.pdf</URL>
       <LINKTITLE>White paper</LINKTITLE>
    </RELATEDLINK>
    <PUBLISHINFO>
          <PUBLISHDIR>/web/netfinity/</PUBLISHDIR>
          <PUBLISHFILENAME>index.html</PUBLISHFILENAME>
          <STYLESHEET>web_product_index.xsl</STYLESHEET>
    </PUBLISHINFO>
    <PUBLISHINFO>

          <PUBLISHDIR>/pda/netfinity/</PUBLISHDIR>
          <PUBLISHFILENAME>index.html</PUBLISHFILENAME>
          <STYLESHEET>pda_product_index.xsl</STYLESHEET>
    </PUBLISHINFO>
    <GROUPINDEX>444-ifrg.xml</GROUPINDEX>
    <MAINPHOTO SUBFRAGMENTTYPE="IMAGEFRAGMENT">
          222-bfrg.xml</MAINPHOTO>
    <HIGHLIGHTS SUBFRAGMENTTYPE="LISTFRAGMENT">
          444-tfrg.xml</HIGHLIGHTS>
</PRODUCTPAGE>
```

Once all DTDs for the collection have been defined, save them in the directory defined by the variable *dtdDir* in the *franklin.properties* file.

After updating, adding or deleting any DTDs, update the files *configDir/dtds.xml* and *dtdDir/subfragmenttypes.txt* to reflect the current DTDs. Also remember to define a DAD file to for each DTD. (**Future**: DAD explanation should be expanded)

Step 7: Define Style Sheets

For each servable DTD, you need to define one or more XSL style sheets that will be assembled with the servable XML and the XML of any subfragments into the final published pages. A style sheet is written using the XSL syntax to produce HTML, DHTML, HDML or other desired output. [See the XSL Transformations syntax at http://www.w3.org/TR/xslt]

Comment [LW3]: mention LotusXSL and have url to alphaworks site.

Franklin specification of XSL style sheets

Because the servable includes content from subfragments, the style sheet must be written to work on the so-called *expanded* servable. Before page assembly, a servable is temporarily rewritten to include the content of all its subfragments. Because the XLink standard has not been finalized,

XSL style sheets cannot access content stored in subfragment files outside the servable. Franklin implements a temporary solution that mimics the XLink functionality by expanding the servable. This is demonstrated by the expanded product page 2-tsrv.xml:

```
<?xml version="1.0"?>
<!DOCTYPE PRODUCTPAGE SYSTEM
"http://yourfranklinserver/dtd/productpage.dtd">
<PRODUCTPAGE>
   <SYSTEM>
      <FRAGMENTID>2-tsrv.xml</FRAGMENTID>
      <CREATOR>Joe Moe</CREATOR>
      <MODIFIER>Jane Mane</MODIFIER>
      <CREATIONTIME>384738740383</CREATIONTIME>
      <LASTMODIFIEDTIME>384738740383</LASTMODIFIEDTIME>
      <PAGETYPE>Servable</PAGETYPE>
   </SYSTEM>
   <TITLE>Netfinity 8500R</TITLE>
   <SOURCE>IBM PC Company</SOURCE>
   <SHORTDESCRIPTION>Mainframe features bring extraordinary performance and
reliability to a rack-optimized 8-way server</SHORTDESCRIPTION>
  <LONGDESCRIPTION>A great value in 8-way servers, the new Netfinity 8500R
maximizes uptime and provides superior manageability for compute-intensive
business intelligence, transaction processing and server consolidation
projects. </LONGDESCRIPTION>
   <KEYWORD>New</KEYWORD>
   <KEYWORD>Server</KEYWORD>
   <KEYWORD>Pentium</KEYWORD>
   <CATEGORY>Netfinity 8500R</CATEGORY>
   <CATEGORY>Small and Medium Business</CATEGORY>
   <RELATEDLINK>
      <URL>ftp://ftp.pc.ibm.com/pub/pccbbs/pc_servers/8500rf.pdf</URL>
      <LINKTITLE>White paper</LINKTITLE>
   </RELATEDLINK>
   <PUBLISHINFO>
         <PUBLISHDIR>/web/netfinity/</PUBLISHDIR>
         <PUBLISHFILENAME>index.html</PUBLISHFILENAME>
         <STYLESHEET>web_product_index.xsl</STYLESHEET>
   </PUBLISHINFO>
   <PUBLISHINFO>
         <PUBLISHDIR>/pda/netfinity/</PUBLISHDIR>
         <PUBLISHFILENAME>index.html</PUBLISHFILENAME>
         <STYLESHEET>pda_product_index.xsl</STYLESHEET>
   </PUBLISHINFO>
   <GROUPINDEX>444-ifrg.xml</GROUPINDEX>
   <MAINPHOTO SUBFRAGMENTTYPE="IMAGEFRAGMENT">
    <IMAGEFRAGMENT>
         <SYSTEM>
               <FRAGMENTID>222-bfrg.xml</FRAGMENTID>
               <CREATOR>BOB</CREATOR>
               <MODIFIER>BOB</MODIFIER>
               <CREATIONTIME>384738740383</CREATIONTIME>
               <LASTMODIFIEDTIME>384738740383</LASTMODIFIEDTIME>
               <PAGETYPE>Fragment</PAGETYPE>
               <CONTENTSIZE>400</CONTENTSIZE>
         </SYSTEM>
         <TITLE>The Netfinity 8500R large jpeg</TITLE>
         <SHORTDESCRIPTION>Netfinity 8500R</SHORTDESCRIPTION>
```

```
        <CONTENTDIR>/images/prod_images/</CONTENTDIR>
        <CONTENTFILENAME>8500R_large.jpg</CONTENTFILENAME>
        <CONTENT/>
      </IMAGEFRAGMENT>
    </MAINPHOTO>
    <HIGHLIGHTS SUBFRAGMENTTYPE="LISTFRAGMENT">
       <LISTFRAGMENT>
          <SYSTEM>
                <FRAGMENTID>444-tfrg.xml</FRAGMENTID>
                <CREATOR>BOB</CREATOR>
                <MODIFIER>BOB</MODIFIER>
                <CREATIONTIME>384738740383</CREATIONTIME>
                <PAGETYPE>Fragment</PAGETYPE>
                <LASTMODIFIEDTIME>384738740383</LASTMODIFIEDTIME>
          </SYSTEM>
          <TITLE>Highlights of the 8500R</TITLE>
          <CATEGORY>Netfinity 8500R</CATEGORY>
          <LISTITEM>
                      <TITLE>Light-Path Diagnostics</TITLE>
                      <BODY>Lighted guidance system to assist in quick
identification of failing components, similar to the lights in a copier
that identify the location of a paper jam.</BODY>
          </LISTITEM>
          <LISTITEM>
                      <TITLE>Active PCI technology</TITLE>
                      <BODY>Enables IBM's unique hot-add PCI, letting you
add client systems, balance network traffic or increase storage capacity
without shutting the system down. </BODY>
          </LISTITEM>
       </LISTFRAGMENT>
    </HIGHLIGHTS>
</PRODUCTPAGE>
```

Any Xpath expressions in the style sheet that refer to subfragment content will be local to the servable. The following example XSL style sheet, web_product_index.xsl, produces a simple HTML page by displaying content form the servable as well as the two subfragments:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">


<xsl:template match="PRODUCTPAGE">
<HTML>
<HEAD>
<TITLE><xsl:value-of select="TITLE"/></TITLE>
<META NAME="source-xml" CONTENT="{SYSTEM/FRAGMENTID}"/>
<META NAME="source-xsl" CONTENT="web_product_index.xsl"/>
</HEAD>
<BODY>
<TABLE CELLPADDING="0" CELLSPACING="0" BORDER="0" WIDTH="451">
<TR>
<TD>
<!-- title section -->
    <FONT COLOR="#003399" SIZE="+3" FACE="Times New Roman">
    <xsl:value-of select="TITLE"/>
    </FONT><BR/><BR/>
```

```
<!-- end title section -->
<!-- short description section -->
    <B><FONT SIZE="-1" FACE="Arial">
    <xsl:value-of select="SHORTDESCRIPTION"/>
    </FONT><BR/><BR/></B>
<!-- end short description section -->
</TD>
</TR>
<TR>
<TD>
<!-- photo section -->
    <IMG  HEIGHT="110" WIDTH="140"
SRC="/multimedia{MAINPHOTO/IMAGEFRAGMENT/CONTENTDIR}{MAINPHOTO/IMAGEFRAGMEN
T/CONTENTFILENAME}" BORDER="0"
ALT="MAINPHOTO/IMAGEFRAGMENT/SHORTDESCRIPTION"></IMG>
<!-- end of photo section -->
</TD>
</TR>
</TABLE>
<!-- Highlights section -->
<TABLE BORDER="0" CELLSPACING="0" CELLPADDING="5" WIDTH="451">
<TR>
<TD COLSPAN="2" WIDTH="451">
    <B><FONT SIZE="-1" FACE="Arial">Highlights of the
    <xsl:value-of select="TITLE"/>
    </FONT></B>
</TD>
</TR>
<xsl:apply-templates select="HIGHLIGHTS/LISTFRAGMENT/LIST"/>
</TABLE>
<!-- End of Highlights section -->
</BODY>
</HTML>
</xsl:template>


<xsl:template match="/PRODUCTPAGE/HIGHLIGHTS/LISTFRAGMENT">
<xsl:for-each select="LISTITEM">
    <TR>
    <TD WIDTH="151" VALIGN="TOP">
    <FONT size="-1" face="Arial"><xsl:value-of select="TITLE"/></FONT>
    </TD>
    <TD WIDTH="300" VALIGN="TOP">
    <FONT size="-1" face="Arial"><xsl:value-of select="BODY"/></FONT>
    </TD>
    </TR>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Once all XSL style sheets for the DTDs have been defined, add them to the CHOICES list of the STYLESHEET element in the appropriate DTDs.

Then, check them in to the */xsl* directory using the Franklin Editor.

[Add here, the definition of a debug style sheet, what is should do, and where it should be saved on the server]

Step 8: Create Directory Structure

Create the directory structure that corresponds to the site map of your application. Under *publishDir* create the HTML directories, and under *multimediaDir* the multimedia directories.

If your site is published to more than one audience segment or device, define several *sibling* directory structures under *publishDir*. For example, a site published for two devices, one for browsing all content using a web browser, and another for browsing only the news section using a pda browser, could have the following directory structure:

publishDir/web/
publishDir/web/news/
publishDir/web/products/
publishDir/pda/
publishDir/pda/news/
multimediaDir/images/
multimediaDir/audio/

When an editor authors a servable, the PUBLISHDIR element of the servable will be presented in the UI with the BROWSESERVER widget. This widget allows the editor to browse the directory structure under *assetsDir* and select where to save the resulting file. Similarly, when editing a multimedia object, the widget allows the editor to browse the directory structure to select where to save the binary file.

Step 9: Configure Web Server

To browse the published sites, set up a web server for each sibling site. Configure the *Document Root* variable to point to the top of the directory hierarchy of a sibling site. The example below assumes that the *baseDir* in *franklinServletInitialization.properties* is set to "/franklin/data/":

For the example in the previous section, you would set up two web servers:

```
Web Server 1: DocumentRoot "/franklin/data/publish/web/"
Web Server 2: DocumentRoot "/franklin/data/publish/pda/"
```

Also, add the aliases below to the configuration file of Web Server 1 and 2.

```
Alias /dtd/          "/franklin/data/dtd/"
Alias /xsl/          "/franklin/data/assets/xsl/"
Alias /multimedia/   "/franklin/data/assets/multimedia/"
Alias /xml/          "/franklin/data/xml"
```

Set directory browsing "on" so that you can easily browse the DTDs and verify the uploaded XML files, XSL style sheets, and multimedia files.

Step 10: Define Roles & Users

Before running the Editor UI, you need to define the allowed roles and users of the Franklin system. A role is defined by the list of DTDs the role is allowed to edit. A user is defined by one or more roles.

To define new roles, edit the file *configDir/roles.xml* by importing the DTD *configDir/roles.dtd* to an XML editor such as Xeena from IBM alphaworks. (Future: use the Editor UI to edit the file)

The DTD *roles.dtd:*

```
<!ELEMENT ROLES              (ROLE*)>
<!ELEMENT ROLE               (ROLENAME, DTD*)>
<!ELEMENT ROLENAME           (#PCDATA)>
<!ELEMENT DTD                (#PCDATA)>
```

An example *roles.xml* defines two roles, FragmentEditor and Editor and associates the allowed DTDs to each:

```
<?xml version="1.0" encoding="UTF-8"?>
<ROLES>
    <ROLE>
          <ROLENAME>FragmentEditor</ROLENAME>
          <DTD>http://franklinserver/dtd/textfragment.dtd</DTD>
          <DTD>http://franklinserver/dtd/listfragment.dtd</DTD>
          <DTD>http://franklinserver/dtd/audiofragment.dtd</DTD>
          <DTD>http://franklinserver/dtd/videofragment.dtd</DTD>
          <DTD>http://franklinserver/dtd/imagefragment.dtd</DTD>
    </ROLE>
    <ROLE>
          <ROLENAME>Editor</ROLENAME>
          <DTD>http://franklinserver/dtd/textfragment.dtd</DTD>
          <DTD>http://franklinserver/dtd/newsarticle.dtd</DTD>
          <DTD>http://franklinserver/dtd/productpage.dtd</DTD>
    </ROLE>
</ROLES>
```

To define new users, edit the file *configDir/users.xml* by importing the DTD *configDir/users.dtd* to an XML editor.

The DTD users.dtd:

```
<!ELEMENT USERS              (USER*)>
<!ELEMENT USER               (NAME, EMAIL, PASSWORD, ROLE+)>
<!ELEMENT NAME               (#PCDATA)>
<!ELEMENT EMAIL              (#PCDATA)>
<!ELEMENT PASSWORD           (#PCDATA)>
<!ELEMENT ROLE               (#PCDATA)>
```

An example *users.xml* defines two users, each with one or more roles.

```
<?xml version="1.0" encoding="UTF-8"?>
<USERS>
```

```
    <USER>
     <NAME>Joe Moe</NAME>
     <EMAIL>joe@us.ibm.com</EMAIL>
     <PASSWORD>joe</PASSWORD>
     <ROLE>FragmentEditor</ROLE>
     <ROLE>Editor</ROLE>

    </USER>
    <USER>
      <NAME>Jane Mane</NAME>
      <EMAIL>jane@us.ibm.com</EMAIL>
      <PASSWORD>jane</PASSWORD>
      <ROLE>Editor</ROLE>
     </USER>
    <USER>
</USERS>
```

When the Franklin server is initialized, the Dispatcher module runs *Globals.loadinfo()*. This method merges *users.xml*, *roles.xml* and *dtds.xml* into one DOM in memory for fast access. The method verifies that all roles named in *users.xml* have a definition in *roles.xml*. It also verifies that all DTDs named in *roles.xml* are defined in *dtds.xml* and exist in the named directory. If any discrepancies are detected, the server prints out a warning message. (Future: the verification still needs to be implemented.)

(Future: if any of the configuration files have changed after the server was last initialized, the files will get reloaded and the DOM will get refreshed. This will not have an effect on any users currently logged on.)

Editor Interface & Dispatcher Communication

After installing the Editor User Interface application and completing the customization described in Section Install Franklin Client, launch the application from the Franklin Editor icon on the desktop.

All communications between the Editor UI and the Franklin Dispatcher follow the WebDAV protocol. [See the full specification at http://www.webdav.org/specs/]

The HTTP header of a Client request always contains the following attributes with *ACTION* replaced by PUT, GET, LOCK or UNLOCK, *franklinservername* replaced by the name of the Franklin server the client is communicating with, and *length* replaced by the length in bytes of the body section.

```
ACTION /filename HTTP/1.1
Host: franklinservername
Content-type: text/xml; charset="utf-8"
Content-Length: length
```

The body of the Client request contains the XML document to be checked into the server or a DASL search query.

The HTTP header of the Dispatcher response always contains the following attributes with *errorcode* and *message* replaced by the standard codes listed in Appendix 1.

```
HTTP/1.1 errorcode message
Content-Type: text/xml; charset="utf-8"
Content-Length: length
```

The format of the body of the Dispatcher response depends on the request and whether the request was successfully completed or not.

A special format used for the response follows the DTD below. In the further examples, the use of the elements will become obvious.

```
<!ELEMENT RESPONSE (PROCESS, STATUS, ERRORCODE, MESSAGE, SYSTEM?, LOCK?)>
<!ELEMENT PROCESS              (#PCDATA)>
<!ELEMENT STATUS               (#PCDATA)>
<!ELEMENT ERRORCODE            (#PCDATA)>
<!ELEMENT MESSAGE              (#PCDATA)>
<!ELEMENT SYSTEM           (FRAGMENTID, CREATOR, MODIFIER, CREATIONTIME,
LASTMODIFIEDTIME, PAGETYPE, CONTENTSIZE?)>
<!ELEMENT FRAGMENTID       (#PCDATA)>
<!ELEMENT CREATOR          (#PCDATA)>
<!ELEMENT MODIFIER         (#PCDATA)>
<!ELEMENT CREATIONTIME     (#PCDATA)>
<!ELEMENT LASTMODIFIEDTIME (#PCDATA)>
<!ELEMENT PAGETYPE         (#PCDATA)>
<!ELEMENT CONTENTSIZE      (#PCDATA)>
<!ELEMENT LOCK             (LOCKEDBY, LOCKTIME, LOCKTOKEN?)>
<!ELEMENT LOCKEDBY         (#PCDATA)>
<!ELEMENT LOCKTIME         (#PCDATA)>
<!ELEMENT LOCKTOKEN        (#PCDATA)>
```

Login

The editor logs in using the user name and password defined by the Franklin administrator, as defined in Section Define Roles & Users.

Client request:

```
GET /xml/franklin_init.xml HTTP1.1
Host: franklinserver/franklinservlet
Content-Type: text/xml; charset="utf-8"

Authorization: "Basic " + encode Base64(username + ":" + password)
```

If the user name is not defined or if the password is entered incorrectly, the dispatcher responds with the appropriate error. Dispatcher response:

```
HTTP/1.1 401 SC_UNAUTHORIZED
Content-Type: text/xml; charset="utf-8"
Content-Length: length

<?xml version="1.0"?>
<RESPONSE>
  <PROCESS>login</PROCESS>
  <STATUS>401</STATUS>
```

Comment [LW4]: I don't think this is the right error code returned here. But couldn't tell cus its not working

```
  <ERRORCODE>U101</ERRORCODE>
  <MESSAGE>User Joe Doe not defined</MESSAGE>
</RESPONSE>
```

If login succeeds, as described in Section Dispatcher: Session Management, the Dispatcher adds the user to the *currentusers* hash table and generates a unique session identifier, *sessionId.* All subsequent requests from the Editor UI must contain *sessionId* in the HTTP header.

The dispatcher responds to a successful login request by generating the *franklin_init.xml* document that corresponds to the user's roles, references the DTDs the user is allowed to edit, and specifies the attributes, operators and allowed values for the Search UI. Dispatcher response:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: length
Sessionid: 175a:dc8e0de306:-8000

<?xml version="1.0" encoding="UTF-8"?>
<FRANKLIN_INIT>
   <SEARCH>
     <ATTRIBUTELIST>
          <ATTRIBUTE displayname="Creation Date" name="CREATIONTIME"
class="Time"/>
          <ATTRIBUTE displayname="Last Modified Date"
name="LASTMODIFIEDTIME" class="Time"/>
          <ATTRIBUTE displayname="Creator" name="CREATOR" class="Name"/>
          <ATTRIBUTE displayname="Document Type" name="DOCTYPE"
class="Selection" options="TEXTFRAGMENT | LISTFRAGMENT | IMAGEFRAGMENT |
AUDIOFRAGMENT | VIDEOFRAGMENT | INDEXGROUP | PRODUCTPAGE | SOMESERVABLE"/>
          <ATTRIBUTE displayname="Page Type" name="PAGETYPE"
class="Selection" options="Fragment|Servable"/>
          <ATTRIBUTE displayname="Keyword" name="KEYWORD" class="Text"/>
          <ATTRIBUTE displayname="Category" name="CATEGORY"
lass="Selection" options=" Netfinity_8500R | Netfinity_7000_M10 |
Netfinity_5500_M10 | Netfinity_5600 | Netfinity_5500"/>
   </ATTRIBUTELIST>
   <CLASSLIST>
         <CLASS name="Time">
               <OPERATOR name=">="/>
               <OPERATOR name="&#60;="/>
               <OPERATOR name="="/>
               <VALUE datatype="date"/>
         </CLASS>
         <CLASS name="Integer">
               <OPERATOR name=">="/>
               <OPERATOR name="&#60;="/>
               <OPERATOR name="="/>
               <VALUE datatype="integer"/>
         </CLASS>
         <CLASS name="Name">
               <OPERATOR name="is"/>
               <OPERATOR name="isn't"/>
               <OPERATOR name="starts with"/>
               <VALUE datatype="string"/>
         </CLASS>
```

```
            <CLASS name="Text">
                  <OPERATOR name="is"/>
                  <OPERATOR name="starts with"/>
                  <VALUE datatype="string"/>
            </CLASS>
            <CLASS name="Selection">
                  <OPERATOR name="is"/>
                  <OPERATOR name="isn't"/>
                  <VALUE datatype="choice"/>
            </CLASS>
      </CLASSLIST>
      <RESULTS>
      <ATTRIBUTE displayname="Last Modified Date" name="LASTMODIFIEDTIME"
class="Time"/>
      <ATTRIBUTE displayname="Creator" name="CREATOR" class="Name"/>
      <ATTRIBUTE displayname="Title" name="TITLE" class="Text"/>
      <ATTRIBUTE displayname="Document Type" name="DOCTYPE"
lass="Selection"/>
      </RESULTS>
    </SEARCH>
   <ROLE name="FragmentEditor" displayname="Fragment Editor">
      <FRAGMENTS displayname="Fragment">
            <DTD displayname="Text"
href="http://franklinserver/franklin/dtd/textfragment.dtd"/>
            <DTD displayname="List"
href="http://franklinserver/franklin/dtd/listfragment.dtd"/>
            <DTD displayname="Audio"
href="http://franklinserver/franklin/dtd/audiofragment.dtd"/>
            <DTD displayname="Video"
href="http://franklinserver/franklin/dtd/videofragment.dtd"/>
            <DTD displayname="Image"
href="http://franklinserver/franklin/dtd/imagefragment.dtd"/>
      </FRAGMENTS>
      </ROLE>
    <ROLE name="Editor" displayname="Editor">
      <FRAGMENTS displayname="Fragment">
            <DTD displayname="Text"
href="http://franklinserver/franklin/dtd/textfragment.dtd"/>
      </FRAGMENTS>
      <SERVABLES displayname="Page">
            <DTD displayname="News Article"
href="http://franklinserver/franklin/dtd/newsarticle.dtd"/>
           <DTD displayname="Product Page"
href="http://franklinserver/franklin/dtd/productpage.dtd"/>
       </SERVABLES>
     </ROLE>
</FRANKLIN_INIT>
```

From this *franklin_init.xml* document, the Editor UI builds the *File > New Fragment* and *File >New Page* menus. It also maintains the mappings between display names and DTD URLs in a hash table.

The Editor UI can be set to load all DTDs at this point, if it is important to enable off-line editing. We have chosen to load a DTD from the server upon demand for a faster startup process.

At this point, the editor is able to either create new content or search for existing content in the system.

Create new content

The *File > New Fragment* menu lists all fragments and the *File > New Page* menu lists all servable pages the editor is allowed to create or edit. If the editor selects to create a new fragment, e.g. a TEXTFRAGMENT, the Editor UI gets the DTD from the appropriate URL and automatically generates a template from the DTD, as shown below:

[Include screenshot here]

In parallel, the Editor UI maintains in memory a DOM corresponding to the DTD.

Editor UI Widgets

The Editor UI uses the DATATYPE attributes in the DTD to generate the appropriate Java widget in the user interface. If an element does not contain a DATATYPE attribute no input is allowed for that element. Children elements may still contain DATATYPE attributes to specify their user interface. The mapping between Franklin UITYPEs and Java widgets are given below:

```
date            => JtextField
integer         => JtextField
string          => JtextField
shorttext       => JtextArea (scrolling)
longtext        => JtextArea (scrolling)
choice          => JComboBox(with DefaultComboBoxModel to hold the data)
browselocal     => JtextField(with JfileChooser to select local file)
browseserver    => JtextField(with custom JFrame to browse server
                   directory)
```

Each Java widget is encapsulated in a set of classes that include additional functionality. For example, if an element in the DTD is required, e.g. TITLE, the widget will be highlighted (e.g. colored brightly) to help the editor distinguish which fields must be filled in. If an element can appear more than once, e.g. KEYWORD, +/- buttons appear next to the widget that allow duplicating the widget or group of widgets.

BODY tags are handled specially within the system. The system assumes that BODY tags are composed of 1 or more PARAGRAPH tags. Typically, this is represented by a longtext widget in the user interface. Blank lines in the input are interpreted as paragraph separators. When constructing the DOM object, these paragraphs are composed within the outer BODY tag.

Check-in of New Fragment

When the editor has filled in the template in the UI, clicking on the check-in icon verifies that all required elements are filled in. If so, the DOM in memory is populated with the data in the UI widgets. New nodes are added if new instances of an element have been created using the +/- buttons. Nodes are removed from the DOM if optional fields have not been filled in. Once the DOM mirrors exactly the UI, the document is transformed to an XML string and a request is sent to the Dispatcher with the XML as the body.

```
    <LASTMODIFIEDTIME>                              /LASTMODIFIEDTIME>
    <PAGETYPE>Fragment</PAGETYPE>
    <CONTENTSIZE>537</CONTENTSIZE>
  </SYSTEM>
</RESPONSE>
```

If the fragment about to be checked in has an accompanying content file, i.e. a multimedia asset or an XSL style sheet, the Editor UI encodes the contents using Base64encoding into the CONTENT element in the XML. On the server side, the Dispatcher un-encodes it and writes the file to the file system, as described in Section XXX. This method avoids sending multiple requests to the Dispatcher and having to maintain state between two requests.
Caveat: presently we are unclear on whether there is a size limitation to this method! And it is slow!

Check-In of Modified Fragment

If the Editor UI checks in a modified fragment or page, it will have received a LOCKTOKEN from the Dispatcher before checking it out. The check-in request in this case must include the LockTocken header field.

Deleted:

Client Request

```
PUT fragmentid HTTP1.1
Host: franklinserver/franklinservlet
Content-Type: text/xml; charset="utf-8"
SessionId: 175a:dc8e0de306:-8000
Content-Length: length
LockTocken:
```

The dispatcher verifies that the correct lock token is supplied with the PUT request before it processes the request. The dispatcher changes the MODIFIER and the LASTMODIFIEDTIME fields in the SYSTEM element.

After the check-in command, the Editor UI issues an UNLOCK command using the appropriate LOCKTOKEN.

Client request:

```
UNLOCK /12345678-tfrg.xml HTTP1.1
Host: franklinserver/franklinservlet
Sessionid  = 175a:dc8e0de306:-8000
Locktocken = 12345678
```

The Dispatcher verifies the LOCKTOKEN as described in Section XXX, and returns an OK if the token is correct, otherwise it sends one of the two Franklin lock errors:
L101 = Lock tokens do not match
L102 = Missing lock token

Check-out

To check out a fragment for editing from the Franklin Server, the Editor UI first requests a lock for the given fragment, as defined by the WebDAV protocol.

Client request:

```
LOCK /12345678-tfrg.xml HTTP1.1
Host: franklinserver/franklinservlet
Sessionid  = 175a:dc8e0de306:-8000
```

If the fragment is already locked by another user, the dispatcher returns a response with information on who has locked it when, in case the user wants to contact the person who holds the lock.

Dispatcher response:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: length
SessionId: 175a:dc8e0de306:-8000

<?xml version="1.0"?>
<RESPONSE>
  <PROCESS>Lock</PROCESS>
  <STATUS>200</STATUS>
  <ERRORCODE>OK</ERRORCODE>
  <MESSAGE>Locked</MESSAGE>
  <LOCK>
    <LOCKEDBY>Jane Mane</LOCKEDBY>
    <LOCKTIME>                              'LOCKTIME>
  </LOCK>
</RESPONSE>
```

Comment [DM5]: Should not be 200 (I'll have to check)

If the fragment is not already locked and if the user with the *sessionId* is allowed to edit documents based on the DTD of the requested fragment, the dispatcher creates a unique lock on the fragment as described in section Dispatcher: Lock Management. It also sends the lock token back in the response.

Dispatcher response:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: length
SessionId: 175a:dc8e0de306:-8000

<?xml version="1.0"?>
<RESPONSE>
  <PROCESS>Lock</PROCESS>
  <STATUS>200</STATUS>
  <ERRORCODE>OK</ERRORCODE>
  <MESSAGE>OK</MESSAGE>
  <LOCK>
    <LOCKEDBY>Joe Moe</LOCKEDBY>
    <LOCKTIME>                              /LOCKTIME>
    <LOCKTOKEN>12345678</LOCKTOCKEN>
```

```
  </LOCK>
</RESPONSE>
```

Now the Editor UI can request the fragment for editing using the lock received from the server.

Client request:

```
GET /12345678-tfrg.xml HTTP1.1
Host: franklinserver/franklinservlet
Content-Type: text/xml; charset="utf-8"
Locktocken = 12345678
Sessionid  = 175a:dc8e0de306:-8000
```

The Dispatcher compares the lock to the one saved in the Meta Data Store. If they match, Dispatcher responds by sending back the complete XML of the fragment.

Dispatcher response:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: length
Sessionid: 175a:dc8e0de306:-8000

 <?xml version="1.0"?>
<!DOCTYPE PRODUCTPAGE SYSTEM "http://franklinserver/dtd/productpage.dtd">
<PRODUCTPAGE>
   <SYSTEM>
     ...
   </SYSTEM>
   ...
</PRODUCTPAGE>
```

The Editor UI retrieves the DTD of the checked-out fragment from the server if it has not yet loaded it. Using the DTD it auto-generates the UI widgets and then fills in the existing values from the checked-out fragment, adding new instances of elements using the +/- mechanism when necessary.

The editor can modify the fields in the same way as when creating new content. Upon check-in the Dispatcher updates the LASTMODIFEDTIME and MODIFIER fields in the SYSTEM data of the checked-in XML document.

Search

Clicking on the Search icon in the Editor UI brings up the Search dialogue shown below:

[insert Search screen shot here]

When launched, the Search dialogue parses *franklin_init.xml* and stores attributes, operators and allowed values into hash tables. It dynamically generates the widgets for the query composition. When new attributes or values are added to *franklin_init.xml*, the Search code does not need to be updated.

The Search UI communicates with the Dispatcher using the Distributed Authoring Search Language (DASL) [add ref], an extension to the WebDAV protocol. Franklin Server defines the

"Franklin" name space which allows the insertion of properties that correspond to the names of the indexed elements in Franklin Meta Data Store.

An example of a DASL exchange between Search UI and Dispatcher is shown below for the example Boolean query:

```
(AND
        (PAGETYPE "is" "Fragment")
        (LASTMODIFIEDTIME "gte" "
        (CREATOR "is not" "Jeff Milton"))
```

Search request:

```
SEARCH / HTTP1.1
Host: franklinserver/franklinservlet
Content-Type: text/xml; charset="utf-8"
Sessionid  = 175a:dc8e0de306:-8000

<?xml version="1.0"?>
<d:searchrequest xmlns:d="DAV:" xmlns:f="Franklin:">
  <d:basicsearch>
    <d:select>
      <d:prop>
         <f:FRAGMENTID/>
          <f:DTD/>
         <f:LASTMODIFIEDTIME/>
         <f:TITLE/>
         <f:CREATOR/>
      </d:prop>
    </d:select>
    <d:from>
      <d:scope>
        <d:href/>
        <d:depth>infinity</d:depth>
      </d:scope>
    </d:from>
    <d:where>
      <d:and>
          <d:eq>
            <d:prop> <f:PAGETYPE/> </D:prop>
            <d:literal>Fragment</D:literal>
        </d:eq>
          <d:gte>
             <d:prop> <f:LASTMODIFIEDTIME/> </D:prop>
             <d:literal>1999-10-10</D:literal>
          </d:gte>
          <d:not>
             <d:eq>
                <d:prop> <f:CREATOR/> </D:prop>
                <d:literal>Jeff Milton</D:literal>
             </d:eq>
       </d:not>
     </d:and>
    </d:where>
    <d:limit>
```

```
      <d:nresults>2</d:nresults>
      </d:limit>
    </d:basicsearch>
  </d:searchrequest>
```

The Dispatcher passes the query to the Meta Data Store where the query is converted to SQL and executed against DB2. The results are converted back to the DASL format.

Currently, we are using the d:nresults tag to indicate a range of results to be returned. (e.g. <d:nresults>1-50</d:nresults>) This enables the client to request subsequent "pages" from a search with a large number of results.

Dispatcher response:

```
HTTP/1.1 207 Multi-Status
Content-Type: text/xml; charset="utf-8"
Content-Length: length

<?xml version="1.0"?>
<d:multistatus xmlns:d="DAV:" xmlns:f="Franklin:">
  <f:responsesummary>
    <f:start>1</f:start>
    <f:end>2</f:end>
    <f:total>45</f:total>
  </f:responsesummary>
  <d:response>
    <d:href>http://franklin.adtech.ibm.com/43987548-tfrg.xml</d:href>
    <d:propstat>
          <d:prop>
                <f:FRAGMENTID>43987548-tfrg.xml</f:FRAGMENTID>
                <f:DTD>Textfragment</f:DTD>
                <f:PAGETYPE>Fragment</f:PAGETYPE>
                <f:LASTMODIFIEDTIME>
               </f:LASTMODIFIEDTIME>
                <f:TITLE>Lou Gerstner's bio</f:TITLE>
                <f:CREATOR/>Joe Moe</f:CREATOR>
          </d:prop>
    </d:propstat>
 </d:response>
<d:response>
    <d:href>http://franklin.adtech.ibm.com/9999999-frg.xml</d:href>
    <d:propstat>
          <d:prop>
          <f:FRAGMENTID>9999999-tfrg.xml</f:FRAGMENTID>
                <f:DTD>Listfragment</f:DTD>
                <f:PAGETYPE>Fragment</f:PAGETYPE>
                <f:LASTMODIFIEDTIME>
               </f:LASTMODIFIEDTIME>
                <f:TITLE>Highlights for Netfinity 8500R</f:TITLE>
                <f:CREATOR/>Jane Mane</f:CREATOR>
          </d:prop>
    </d:propstat>
 </d:response>
</d:multistatus>
```

Comment [LW6]: This example is not quite right (ie. the numbers aren't specified this way, I don't think)

The Search UI parses the results and displays them in the results table. From the table, the editor can select items and merge them into the Active List in the Editor UI.

Future: If more than the requested number of hits exist in the database, the Search UI uses the `RESPONSESUMMARY` element in the result list to determine how to manage the "Next" and "Previous" buttons that allow further results or to go back to a previous set of results.

Preview

Before checking in a servable, the editor can preview the final page to be published. The preview icon in the Editor UI is active only when editing a servable. Requesting a preview sends a request to the Preview Manager servlet with the temporary contents of the servable XML. The Preview Manager expands the servable with contents of any subfragments, assembles the page with all included style sheets using LotusXSL, and returns the resulting HTML output to the client.

The Editor UI launches the web browser specified in the *franklin.properties* file and displays the temporary output. Once satisfied, the editor can check in the servable. Servables previously checked in (e.g., servables returned as search results) can also be previewed.

(Future: preview of an HTML page does not indicate to the editor which fragment produced any given area of the page. Need to devise a way to display the source element.)

Dispatcher

Session Management

When a user logs on using the Editor UI, as described in the Section Editor Interface & Dispatcher Communication: Login, the dispatcher checks for a valid user and password by consulting the DOM, generated at startup, which contains all user information. If they are valid, the dispatcher adds the user to the *currentusers* hash table and generates a unique session identifier, *sessionId*. *SessionId* is created using the java *UID()* call that guarantees to return a unique identifier for the machine the process is running on.

If the same user logs on a second time from another terminal without terminating the earlier session, the earlier *sessionId* in overwritten by a new one, and the old session becomes invalid.

At logout, the users entry is removed from the *currentusers* hash table.

Future: the class that manages the user sessions must be serializable, so that its state can be saved and reloaded if the Franklin Server servlet needs to be restarted.

System Data Creation

When a dispatcher receives the check in request from the Editor UI, it handles new and modified fragments differently.

For a new fragment, the Dispatcher fills in the following so-called SYSTEM elements in the DOM built from the incoming fragment:

```
<SYSTEM>
  <FRAGMENTID>46ba850dc8cf1f3c1007f33-tfrg.xml</FRAGMENTID>
  <CREATOR>Joe Doe</CREATOR>
  <MODIFIER>Joe Doe</MODIFIER>
  <CREATIONTIME>                                  /CREATIONTIME>
  <LASTMODIFIEDTIME>                                  LASTMODIFIEDTIME>
  <PAGETYPE>Fragment</PAGETYPE>
  <CONTENTSIZE>537</CONTENTSIZE>
</SYSTEM>
```

FRAGMENTID is the unique identifier for the fragment. This identifier is created using the java *UID()* call which returns a guaranteed unique identifier for the machine where the process is running. To the UID, the dispatcher appends the suffix *–tfrg.xml* for simple fragments, *-bfrg.xml* for compound fragments, or *-tsrv.xml* for servables. To know which suffix to append, the dispatcher consults the *dtdToType* hash table, built at startup to cache the mapping between DTDs and their types.

CREATOR is the name of the user who originally checked in the new fragment. The dispatcher gets this name by calling the *sessionidToUser* method, which retrieves the user name from the *currentusers* hash table based on the *sessionId.*

MODIFIER is the name of the user who is currently checking in the fragment. MODIFIER and CREATOR are the same when creating the system data for a new fragment.

CREATIONTIME is the Java generated time stamp of the system data creation time at original check-in.

LASTMODIFIEDTIME is the Java generated time stamp of the system data update time at subsequent check-ins. CREATIONTIME and LASTMODIFIEDTIME are the same for a new fragment.

PAGETYPE is set to either "FRAGMENT" or "SERVABLE". The dispatcher sets PAGETYPE by consulting the *dtdToType* hash table, built at startup to cache the mapping between DTDs and their types. This field is important because processing of fragments and servables is different in the Editor UI as well as in the content store module.

CONTENTSIZE is the size in bytes of the checked-in fragment including any included binary data. The dispatcher calculates this after filling in the system data but before extracting any binary data. Thus, this is the size of the string being sent over the network between Editor UI and the dispatcher.

Name Space Management

The name space manager module of the dispatcher manages all reading and writing of files. It abstracts away the actual file system from all other modules, so that they do not have to keep track of specialized directories.

The name space manager provides the functionality to read and write into the file system DOMs, corresponding XML strings that represent fragments or servables. At dispatcher startup, the initialization file is read in, and the variables defining the directories become available to the name space manager.

When writing a compound fragment, the name space manager also extracts any encoded multimedia files and style sheets and writes them into the appropriate directories. On the other hand, when reading a compound fragment, it encodes any external files into the XML and returns the DOM to the module requesting it.

In addition to the dispatcher, the content store uses the name space manager as well. The content store uses it to read fragments from the file system and to write HTML/HDML/DHTML output files from page assembly into the file system.

The advantage of separating the name space manager from the rest of the Franklin server is to isolate the knowledge about dedicated file system directories to one module. For example, if *xmlDir* needed to be split up into a series of second-level directories to limit the size of any one directory, only the name space manager would need to know about the change. Under *xmlDir* could reside 10 child directories 0/, 1/, ...9/ and a fragment would be stored in one of them based on a load-balancing algorithm handled by the name space manager.

Coordination Between Modules at Check-in

[describe the 3 phase save to file system, meta store and tm, with the fact that tm is asynchronous. Maintenance of pending jobs table by dispatcher, and roll-back]

Lock Management

As described in the sections Editor Interface & Dispatcher Communication: Check-in and Check-out, the Editor UI and dispatcher exchange lock tokens during the LOCK and UNLOCK requests from the Editor UI.

When the dispatcher issues a lock token, it uses the Java *UID()* call to create a unique identifier. It sends the token along with the user name and the lock time to the Editor UI in the body of the response and stores another copy of this information in the meta-data store as described in the section Meta Data Store: Lock Management.

When the dispatcher receives an UNLOCK request with a lock token from the Editor UI, it needs to verify that the token matches the one stored in the meta-data store. If they match, the dispatcher issues a call to delete the lock in the meta-data store.

The dispatcher has two other ways to manage locks if problems occur. If the Editor UI requests that all locks held by the current user be released, the dispatcher issues the call *releaseLockByUser* to the meta-data store. When the dispatcher is restarted due to a system crash, all pending locks in the meta-data store can be released at startup with the call *releaseLockAll*.

Error Handling

All Franklin server side components abide to the same Franklin error handling scheme. When any of the components called by dispatcher, namely meta-data store, name space manager, user manager, or content manager, catch or throw a Java exception, they convert it to a Franklin exception and fill in all details about the context and the conditions where the error occurred.

A Franklin exception contains the following attributes:

myError - the Franklin error code
myHttpError - the HTTP error code corresponding to the Franklin error code
myMessage – explanation of error, presented to user of the Client application
myDestination – one of ERROR_USER, ERROR_LOG, ERROR_ADMIN to indicate where the error should be directed
myException – the originally caught exception, if there is one

The dispatcher module routes the exception based on the attributes. If *myDestination* is set to ERROR_USER, the dispatcher returns the exception to the Editor UI which displays the error to the user. If it is set to ERROR_LOG, the error is written to the error log file, and for ERROR_ADMIN, the process notifies the system administrator.

Meta Data Store

The meta-data store allows the indexing and searching of fragments and servables. All or a subset of the XML elements can be set to be indexed. For a large content site this allows users to quickly locate content objects of interest. The meta-data store also manages the lock information of content objects.

DB2 XML Extenders

Franklin uses XML Extenders for DB2 to index a subset of the XML elements of fragments and servables. To accomplish indexing, XML Extenders uses a Document Access Definition (DAD) that maps an XML element to a column in a DB2 table.

DB2 XML Extenders provides two different methods, namely XColumn and XCollection. We have implemented both methods in Franklin and decribe both in this document. We recommend using the XCollection as it is more flexible.

XColumn

The current XColumn implementation of XML Extenders can only map one DTD to one or more DB2 tables. In order to map all Franklin DTDs to one or more common tables, the dispatcher converts all DTDs to a so-called *universal* DTD, which contains all elements to be indexed in the set of DTDs. For this universal DTD, two DADs are created based on the XML Extenders syntax.

Two DADs are needed, because the current XColumn implementation does not support inserting elements that occur only once in the XML and those that occur more than once using a single

DAD. Thus, the examples in this section show two DADs that map values from the universal DTD.

When designing the DADs, all elements that appear only once, or single-occurrence elements, can be mapped to one table. Any elements that can appear more than once, or multi-occurrence elements, need to be mapped each into separate dedicated tables. The administrator creates a view between the single-occurrence table and the multi-occurrence tables to perform searches across all tables with one command.

A DAD specifies the following items:

```
table name = name of the DB2 table
column name = name of the column in the encapsulating table
column type = data type of the column
column path = XPath expression from the root to the element to be indexed
in the column
column multi-occurrence = flag to indicate whether the element at the path
can occur more than once
```

Example of a DAD mapping single-occurrence elements:

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "c:\dxx\franklin\dtd\universal.dtd">
<DAD>
   <dtdid>UNIVERSALDTD</dtdid>
   <validation>NO</validation>
<Xcolumn>
      <table name="META.MAIN">
      <column name="CREATOR"
        type="varchar(50)"
        path="/UNIVERSAL/SYSTEM/CREATOR"
        multi_occurrence="NO">
      </column>
      <column name="CREATIONTIME"
        type="TIMESTAMP"
        path="/UNIVERSAL/SYSTEM/CREATIONTIME"
        multi_occurrence="NO">
      </column>
      <column name="LASTMODIFIEDTIME"
        type="TIMESTAMP"
        path="/UNIVERSAL/SYSTEM/LASTMODIFIEDTIME"
        multi_occurrence="NO">
      </column>
      <column name="PAGETYPE"
        type="char(10)"
        path="/UNIVERSAL/SYSTEM/PAGETYPE"
        multi_occurrence="NO">
      </column>
      <column name="CONTENTSIZE"
        type="integer"
        path="/UNIVERSAL/SYSTEM/CONTENTSIZE"
        multi_occurrence="NO">
      </column>
      <column name="TITLE"
        type="varchar(250)"
```

```
        path="/UNIVERSAL/TITLE"
        multi_occurrence="NO">
      </column>
      <column name="DOCTYPE"
        type="varchar(32)"
        path="/UNIVERSAL/DOCTYPE"
        multi_occurrence="NO">
      </column>
      <column name="DTDURL"
        type="varchar(512)"
        path="/UNIVERSAL/DTDURL"
        multi_occurrence="NO">
      </column>
    </table>
</Xcolumn>
```

Example of a DAD mapping multi-occurrence elements:

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "c:\dxx\franklin\dtd\universal.dtd">
<DAD>
   <dtdid>UNIVERSALDTD</dtdid>
   <validation>NO</validation>
   <Xcolumn>
     <table name="META.CATEGORY">
       <column name="CATEGORY"
         type="varchar(128)"
         path="/UNIVERSAL/CATEGORY"
         multi_occurrence="YES">
       </column>
     </table>
     <table name="META.KEYWORD">
       <column name="KEYWORD"
         type="varchar(64)"
         path="/UNIVERSAL/KEYWORD"
         multi_occurrence="YES">
       </column>
     </table>
   </Xcolumn>
</DAD>
```

The tables created by these DADs are shown in the Section Table Design.

XCollection

The XCollection implementation of XML Extenders requires one DAD for each DTD to be mapped into DB2. Unlike XColumn, different DTDs can be mapped to the same DB2 tables. Thus, the XCollection implementation does not required documents to be converted to abide to one universal DTD.

[However, the current XCollection also has a few problem. We have implemented a temporary fix into the meta data store until the problems are addressed]

The DAD corresponding to textfragment.dtd is shown below:

[add DAD here]

Table Design

When a DAD is loaded into DB2, the tables and columns specified in it are automatically created. After the tables are created, the administrator needs to add the following items to the database:

- a column named ISCOMMIT in the table storing the single-occurrence elements. This column indicates if the fragment has successfully been committed to the content store and file system
- indexing on any columns that will be searched
- a view which combines data from all tables for searching

The database tables created by the previous DAD examples are shown below, along with keys, indexes, and the ISCOMMIT column created by the administrator.

```
Schema:
META:  Used for all the tables used by Franklin
INDEX: Used for all the indexes used by Franklin

Tables Generated by DAD:

META.MAIN: This table contains all the elements that occur at most once in
the input XML document

 Column Name          Data Type        Default     Key          Index

   FRAMENTID             CHAR(56)                    Fk ->     index.fragment
                                                    unifrag1    id
   CREATOR               VARCHAR(50)                           index.creator
   CREATIONTIME          TIMESTAMP                             index.creation
                                                               time
  LASTMODIFIEDTIME       TIMESTAMP                             index.lastmodi
                                                               fiedtime
   CONTENTSIZE           INTEGER
   TITLE                 VARCHAR                               index.title
   PAGETYPE              CHAR(10)                              index.pagesize
   DOCTYPE               VARCHAR(32)                           index.doctype
   DTDURL                                                      index.dtdurl
                       VARCHAR(512)
   ISCOMMIT              INTEGER          0

META.KEYWORD: This table contains the KEYWORD elements associated with a
given FRAGMENTID:

Column Name          Data Type        Default     Key          Index

  FRAMENTID              CHAR(56)                    Fk ->     index.fragment
                                                    unifrag2    ed
```

KEYWORD	VARCHAR(64)		index.keyword

META.CATEGORY: This table contains the CATEGORY elements associated with a given FRAGMENTID.

Column Name	Data Type	Default	key	index
FRAMENTID	CHAR(56)		fk	index.fragment id
CATEGORY	VARCHAR(128)			index.category

UNIFRAG1: This table contains two fields,
Fragmentid - the fragmentid for a given XML file
Fragmentxml - the XML file stored in the file system
The function of this table is to trigger filling META.MAIN when a record is inserted

Column Name	Data Type	Default	key	index
FRAMENTID	CHAR(56)		PK	
FRAGMENTXML	DB2XML.XMLFILE			

UNIFRAG2: This table has the same structure as UNIFRAG1. The function is to trigger filling META.KEYWORD and META.CATEGORY when a record is inserted

Column Name	Data Type	Default	key	index
FRAMENTID	CHAR(56)		PK	
FRAGMENTXML	DB2XML.XMLFILE			

Index

When a fragment or servable is checked in, the dispatcher converts the XML file to abide to the universal DTD for indexing in the XColumn implementation. After the conversion, it sends the universal XML to the meta-data store. In the XCollection implementation, the fragment or servable is sent as is to the meta-data store.

For XCollection, the meta-data store enters a pointer to the file into the two tables named UNITFRAG1 and UNIFRAG2 in the previous example. When the record is entered, a trigger copies the elements specified in the DAD to the appropriate tables. The elements are now ready for searching.

For XColumn ...

Search

When the Search UI sends a DASL search expression, described in the section Editor Interface & Dispatcher Communication: Search, to the dispatcher it passes it directly to the meta-data store. The meta-data store converts it to an SQL expression, executes the SQL query and converts the results to the DASL output format.

An example DASL query:

```
<?xml version="1.0"?>
<d:searchrequest xmlns:d="DAV:" xmlns:f="Franklin:">
  <d:basicsearch>
    <d:select>
      <f:prop>
        <f:FRAGMENTID/>
        <f:DOCTYPE/>
        <f:LASTMODIFIEDTIME/>
        <f:TITLE/>
        <f:CREATOR/>
        <f:PAGETYPE/>
      </f:prop>
    </d:select>
    <d:from>
      <d:scope>
        <d:href/>
        <d:depth>infinity</d:depth>
      </d:scope>
    </d:from>
    <d:where>
      <d:and>
        <d:like>
          <d:prop>
            <f:KEYWORD/>
          </d:prop>
          <d:literal>SERVER</d:literal>
        </d:like>
        <d:like>
          <d:prop>
            <f:PAGETYPE/>
          </d:prop>
          <d:literal>FRAGMENT</d:literal>
        </d:like>
      </d:and>
    </d:where>
    <d:limit>
      <d:nresults>1-50</d:nresults>
    </d:limit>
  </d:basicsearch>
</d:searchrequest>
```

The above DASL query converted to SQL:

```
SELECT DISTINCT fragmentID, doctype, lastModifiedTime, title, creator,
pagetype FROM meta.metaall  where  KEYWORD LIKE 'SERVER%' and PAGETYPE LIKE
'FRAGMENT' and iscommit = 1
```

An example DASL output to above query:

```
<?xml version="1.0"?>
<d:multistatus xmlns:d  ="DAV:" xmlns:f="Franklin:">
  <f:responsesummary>
     <f:start>1</f:start>
     <f:end>1</f:end>
     <f:total>1</f:total>
  </f:responsesummary>/
  <d:response>
      <d:href>http://franklinserver/46b3e60dccbcd84db007777-tfrg.xml
      </d:href>
     <d:propstat>
       <d:prop>
        <f:FRAGMENTID>46b3e60dccbcd84db007777-tfrg.xml</f:FRAGMENTID>
        <f:DOCTYPE>TEXTFRAGMENT</f:DOCTYPE>
        <f:LASTMODIFIEDTIME>                                    /f:LASTMODIFIEDTIME>
        <f:TITLE>Netfinity Highlights</f:TITLE>
        <f:CREATOR>Joe Doe</f:CREATOR>
        <f:PAGETYPE>FRAGMENT</f:PAGETYPE>
       </d:prop>
      </d:propstat>
   </d:response>
</d:multistatus>
```

If the number of results is larger than the result set requested by the Search UI, the meta-data store writes the full results into a cache file and only encodes the requested number into the DASL output. The cache file is named using an expression that encodes the query and the *sessionId* of the user. When the Search UI requests the "Next" set of results for the same query, the meta-data store does not re-execute the query. Instead it consults the cache file and extracts the appropriate next set of results. This caching scheme saves the meta-data store from executing the same query several times if the user if simply navigating within the same result set.

Note that if the contents where to change in DB2, the user does not see the updated results until he re-executes the original query without the "Next" or "Previous" flags.

Lock Management

When the dispatcher receives a LOCK command from the Editor UI, it creates a lock for a fragment or servable and sends the lock to the meta-data store to save in DB2. The lock information, namely LOCKTOKEN, LOCKEDOWNER, and LOCKTIME, is stored in the META.LOCK table of the following format:

Column Name	Data Type	Default	Key	Index
FRAMENTID	CHAR(56)		PK	
LOCKOWNER	VARCHAR(50)			
LOCKTIME	TIMESTAMP			
LOCKTOKEN	VARCHAR(34)			

When the dispatcher receives an UNLOCK command from the Editor UI, it issues the unlock command to the meta-data store. The meta-data store deletes the record from the META.LOCK table.

The Content Store – Daedalus (a.k.a Trigger Monitor)

This section describes how the Franklin project has extended three of the Daedalus handlers to enable the system to manage XML fragments and XSL style sheets. For the full Daedalus API documentation, read http://w3.watson.ibm.com/~challngr/papers/daedalus/index.html.

Daedalus is written in pure Java and implements *handlers* as pre-defined actions performed on the various configurable resources. Flexibility is achieved via Java's dynamic loading abilities, by more sophisticated configuration of the resources used by Daedalus, and through the use of *handler* preprocessing of input data. Most entities defined in a configuration file implement a public Java interface. Users may create their own classes to accomplish localized goals, and specify those classes in the configuration file. This permits run-time flexibility without requiring sophisticated efforts on the part of most users, since default classes are supplied to handle the most common situations.

For Franklin, we have created our own classes to implement three handlers: the Extension Parser, the Dependency Parser, and the Page Assembler.

Extension Parser

Within Franklin, Daedalus manages different types of files differently based on their extensions. Servables, simple, compound, and index fragments, style sheets and multimedia assets are all treated slightly differently in the publishing flow.

The Franklin Extension Parser takes in a name of a fragment, and returns an extension used in the Daedalus configuration files to specify actions to take during the publish process:

```
123445-trfg.xml     => tfrg  (text fragment)
123445-bfrg.xml     => bfrg  (binary wrapper fragment)
123445-ifrg.xml     => ifrg  (index fragment)
123445-tsrv.xml     => tsrv  (text servable fragment)
123445-sfrg.xml     => sfrg  (style sheet wrapper fragment)
web_index.xsl       => xsl   (style sheet)
```

The appropriate behavior for each type of fragment (e.g. source-to-sink, assemble-to-sink) is defined in the Daedalus configuration files. Generally, only servables are assembled to the sink.

Dependency Parser

The Franklin Dependency Parser reads through an XML objects that has been checked in and detects two types of dependencies:

1. Servables and fragments can include subfragments, these get stored as an edge of type "composition" in the Daedalus Object Dependency Graph (ODG).
2. Compound fragments include an associated content file, this dependency gets stored as an edge type "composition" in the ODG.
3. Servables can include style sheets, these get stored as an edge type "stylesheet" in the ODG.

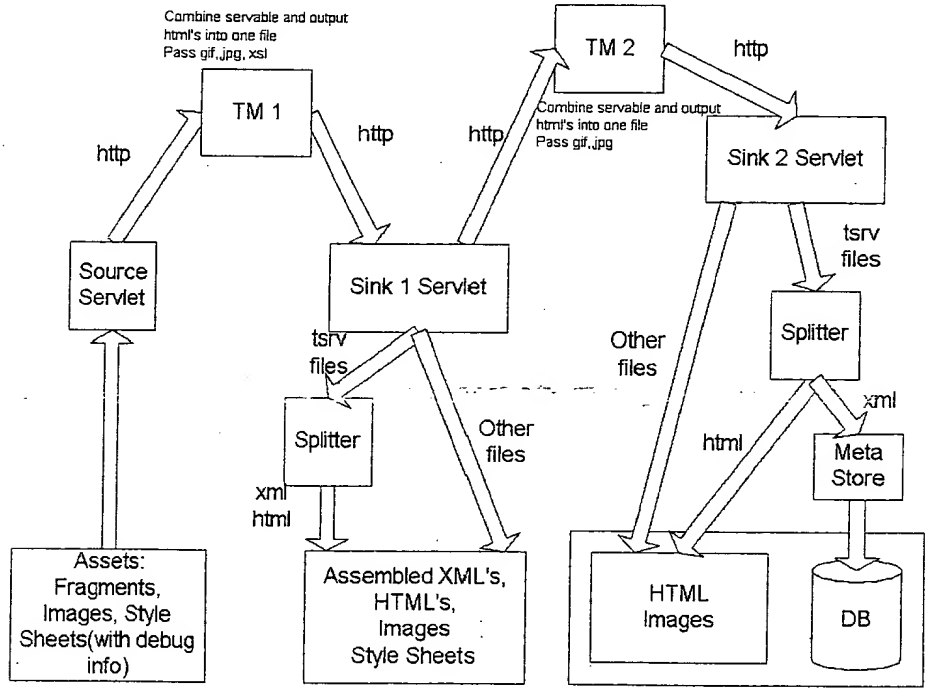
Dependencies are considered to point from the subfragments to the fragments that include them. For binary wrappers, one composition dependency points from the wrapper to the fragment that includes it, and another points from the wrapper to the binary data file that it wraps. For stylesheets, a composition dependency points from the wrapper to the stylesheet, and a stylesheet dependency points from the stylesheet to the servable that uses it.

Page Assembler

The Franklin Page Assembler expands a servable by including the contents of all included subfragments, and combines the resulting XML with the one or more style sheets using LotusXSL to produce HTML output files. The extension of each of the resulting files is determined from the stylesheet names (e.g. web_xxx_**html**.xsl). The assembled XML and all the resulting HTML files are written to one file, which is later split up in the Dispatcher, and the HTML files are written to the appropriate directories in the sink or server.

Chaining of Trigger Monitors

Currently, two Trigger Monitors are used in the publish process. They share an ODG, and the sink of the first one is the source of the second, creating a publishing chain. The following diagram shows the set-up of the Content store in its entirety:



When a fragment is checked in to the Content store, it is added to the shared ODG, and a publish command is issued to the first TM. The TM reads the fragment XML from the source servlet, uses the extension parser to find its extension, and then uses the dependency parser to find dependencies to add to the ODG. The page assembler then pulls in the contents of the fragment's subfragments, and if the fragment is a servable, combines it with its stylesheets to produce the output HTMLs. The servable XMLs, output HTMLs, binary files, and stylesheets are sent to the servlet specified as the sink of the first TM.

When a servable has been approved, a publish command on the servable fragment is issued to the second TM. It is reassembled and recombined with its XSLs, and the resulting XML and HTMLs are published to the second sink servlet. Binary files (such as images) are also published to the second sink. This is where the web server pulls the final HTML and image files from.

Example application

- managing Netfinity pages at ibm.com

Summary

Appendix 1: Error Codes

```
# Status code (101) indicating the server is switching protocols
# according to Upgrade header. (SC_SWITCHING_PROTOCOLS)

X1 = 101

# Status code (200) indicating the request succeeded normally. (SC_OK)

G200 = 200
P200 = 200
OK = 200

# Status code (201) indicating the request succeeded and created
# a new resource on the server. (SC_CREATED)

X3 = 201

# Status code (202) indicating that a request was accepted for
# processing, but was not completed. (SC_ACCEPTED)

X4 = 202

# Status code (203) indicating that the meta information presented
```

```
# by the client did not originate from the server.
(SC_NON_AUTHORITATIVE_INFORMATION)

X5 = 203

# Status code (204) indicating that the request succeeded but that
# there was no new information to return. (SC_NO_CONTENT)

X6 = 204

# Status code (205) indicating that the agent <em>SHOULD</em> reset
# the document view which caused the request to be sent. (SC_RESET_CONTENT)

X7 = 205

# Status code (206) indicating that the server has fulfilled
# the partial GET request for the resource. (SC_PARTIAL_CONTENT)

X8 = 206

# Status code (300) indicating that the requested resource
# corresponds to any one of a set of representations, each with
# its own specific location. (SC_MULTIPLE_CHOICES)

X9 = 300

# Status code (301) indicating that the resource has permanently
# moved to a new location, and that future references should use a
# new URI with their requests. (SC_MOVED_PERMANENTLY)

X10 = 301

# Status code (302) indicating that the resource has temporarily
# moved to another location, but that future references should
# still use the original URI to access the resource. (SC_MOVED_TEMPORARILY)

X11 = 302

# Status code (303) indicating that the response to the request
# can be found under a different URI. (SC_SEE_OTHER)

X12 = 303

# Status code (304) indicating that a conditional GET operation
# found that the resource was available and not modified. (SC_NOT_MODIFIED)
```

```
X13 = 304

# Status code (305) indicating that the requested resource
# <em>MUST</em> be accessed through the proxy given by the
# <code><em>Location</em></code> field. (SC_USE_PROXY)

X14 = 305

# Status code (400) indicating the request sent by the client was
# syntactically incorrect. (SC_BAD_REQUEST)

# THIS IS THE GENERAL (DEFAULT) ERROR RETURNED WHEN ANYTHING BREAKS

#check c102 to make sure it belongs in this area (400)
C101 = 400
C102 = 400
C103 = 400
C123 = 400
C124 = 400
D104 = 400
D110 = 400
P101 = 400
V101 = 400
F100 = 400
F101 = 400
F102 = 400
F103 = 400
F104 = 400
F105 = 400
R101 = 400
R112 = 400
R102 = 400
R103 = 400
R105 = 400
D101 = 400
D111 = 400
D145 = 400
G103 = 400

# Status code (401) indicating that the request requires HTTP
# authentication. (SC_UNAUTHORIZED)

G101 = 401
U101 = 401
U102 = 401
U103 = 401
```

```
L101 = 401
L102 = 401
G104 = 401
# Status code (402) reserved for future use. (SC_PAYMENT_REQUIRED)

X17 = 402

# Status code (403) indicating the server understood the request
# but refused to fulfill it. (SC_FORBIDDEN)

G102 = 403
D123 = 403
# Status code (404) indicating that the requested resource is not
# available. (SC_NOT_FOUND)

X19 = 404

# Status code (405) indicating that the method specified in the
# <code><em>Request-Line</em></code> is not allowed for the resource
# identified by the <code><em>Request-URI</em></code>.
(SC_METHOD_NOT_ALLOWED)

X20 = 405

# Status code (406) indicating that the resource identified by the
# request is only capable of generating response entities which have
# content characteristics not acceptable according to the accept
# headerssent in the request. (SC_NOT_ACCEPTABLE)

F108 = 406

# Status code (407) indicating that the client <em>MUST</em> first
# authenticate itself with the proxy. (SC_PROXY_AUTHENTICATION_REQUIRED)

X22 = 407

# Status code (408) indicating that the client did not produce a
# requestwithin the time that the server was prepared to wait. (SC_REQUEST_TIMEOUT)

X23 = 408

# Status code (409) indicating that the request could not be
# completed due to a conflict with the current state of the
# resource. (SC_CONFLICT)

X24 = 409
```

```
# Status code (410) indicating that the resource is no longer
# available at the server and no forwarding address is known.
# This condition <em>SHOULD</em> be considered permanent. (SC_GONE)

X25 = 410

# Status code (411) indicating that the request cannot be handled
# without a defined <code><em>Content-Length</em></code>. (SC_LENGTH_REQUIRED)

X26 = 411

# Status code (412) indicating that the precondition given in one
# or more of the request-header fields evaluated to false when it
# was tested on the server. (SC_PRECONDITION_FAILED)

X27 = 412

# Status code (413) indicating that the server is refusing to process
# the request because the request entity is larger than the server is
# willing or able to process. (SC_REQUEST_ENTITY_TOO_LARGE)

X28 = 413

# Status code (414) indicating that the server is refusing to service
# the request because the <code><em>Request-URI</em></code> is longer
# than the server is willing to interpret. (SC_REQUEST_URI_TOO_LONG)

X29 = 414

# Status code (415) indicating that the server is refusing to service
# the request because the entity of the request is in a format not
# supported by the requested resource for the requested method.
(SC_UNSUPPORTED_MEDIA_TYPE)

X30 = 415

# Status code (500) indicating an error inside the HTTP server
# which prevented it from fulfilling the request. (SC_INTERNAL_SERVER_ERROR)

X31 = 500

# Status code (501) indicating the HTTP server does not support
# the functionality needed to fulfill the request. (SC_NOT_IMPLEMENTED)

X32 = 501
```

```
# Status code (502) indicating that the HTTP server received an
# invalid response from a server it consulted when acting as a
# proxy or gateway. (SC_BAD_GATEWAY)

X33 = 502

# Status code (503) indicating that the HTTP server is
# temporarily overloaded, and unable to handle the request. (SC_SERVICE_UNAVAILABLE)

X34 = 503

# Status code (504) indicating that the server did not receive
# a timely response from the upstream server while acting as
# a gateway or proxy. (SC_GATEWAY_TIMEOUT)

X35 = 504

# Status code (505) indicating that the server does not support
# or refuses to support the HTTP protocol version that was used
# in the request message. (SC_HTTP_VERSION_NOT_SUPPORTED)

X36 = 505

# Error code in server.dispatcher.Dispatcher

# D104 = Error in Dispatcher.doPost()
# D110 = Fragment Type not Specified or incorrect
# P101 = Error in Dispatcher.putParseRequest()
# V101 = Error validating user

# Error codes in server.Fragment

# F100 = Error in Fragment.fragment2XML()
# F101 = Error opening Fragment.XML2fragment()
# F102 = Error parsing XML file in Fragment.XML2fragment()
# F103 = Error calling readNode("+element+")
# F104 = Error calling getElementValue
# F105 = Error calling getElementType
# F120 = Cannot close StringWriter

# Error codes in server.TextUtils

# R101 = Error in TextFile.read("+filename+")
# R112 = Error in TextFile.readTextFileWOException("+filename+")
# R102 = Error in TextUtils.createDOMfromFile("+xmlfile+")
```

```
# R103 = TextUtils.createDomFromFile SAX exception
# R105 = TextUtils.createDomFromFile IO exception
# R112 = Error in TextFile.readTextFileWOException("+filename+")


# Error codes in server.dispatcher.DomUtils

# D101 = DomUtils.documentToTuniverse TXDOM Exception
# D111 = Error in DomUtils.documentToUniversal
# D123 = Missing or invalid sessionID on checkin

# Error codes in server.dispatcher.Users

# these have destination ERROR_USER

# U101=  User + username + not defined
# U102 = Wrong password  for user + username
# U103 = User with sessionid + sessionId + not defined

# this has destination ERROR_LOG

# U110 = Users.methodname IO exception

# Error codes in server.dispatcher.checkIn

# C103 = Error in document2String
# C102 = Checkin Error
# C101 = Users.checkProvledge error

# DE111 = Delete Error

# G200 = successful get
# P200 = successful put

# Locking errors
# L101 = Lock tokens do not match
# L102 = missing lock token


#MISC
# F108 = invalid FragmentID
# C123 = error in fragment2XML
# C124 = Failed saving content to metadata store

# G104 = Authorization String Empty
# D145 = error parsing input stream
```